www.icgst.com

GreedyHaarSpiker: An Algorithm for In Situ Detection of Highway Lane Boundaries with 1D Haar Wavelet Spikes

Vladimir A. Kulyukin
Department of Computer Science, Utah State University, 4205 Old Main Hill, Logan, UT 84322, USA
vladimir.kulyukin@aggiemail.usu.edu,
http://www.cs.usu.edu/people/VladimirKulyukin

Abstract

An algorithm is presented for in situ vision-based detection of highway lane boundaries on a raspberry pi computer with a raspberry pi camera. The raspberry pi unit is placed inside a Jeep Wrangler, next to the windshield, and is powered through a 12V-to-5V car charger. The algorithm, called *GreedyHaarSpiker*, is based on the detection of 1D Haar Wavelet spikes in 1D Ordered Haar Wavelet Transforms of image rows. To obtain experimental video data for daytime driving, the author drove a 2016 Jeep Wrangler with the installed raspberry pi unit on a sunny day in September 2016 (run 1) and a cloudy day with light rain in November 2016 (run 2) at a speed of 55-60 miles per hour on Route 30, a two-lane Northern Utah highway. To obtain video data of snowy roads and night driving, the author drove the same vehicle on the same highway and at the same speed on a day after a heavy snowfall (run 3) in January 2017 and on the same day after sunset (run 4). Each run was approximately 35 miles long. Each video was segmented into 360 x 240 PNG frames. A sample of 1,000 consecutive frames was selected from each video. The performance of the algorithm was tested on a raspberry pi 3 model B ARMv8 1GB RAM computer on each of the four frame samples. The algorithm is implemented in Python 2.7.9 with OpenCV 3.0. The current implementation processes 20 frames per second.

Keywords: *Computer Vision, Wavelets, Lane Detection, Autonomous Vehicles*

Nomenclature: *CV - Computer Vision, AV - Autonomous Vehicle, HWT - Haar Wavelet Transform*

1 Introduction

Autonomous vehicles (AVs), i.e., vehicles capable of navigating various environments without human input, have featured prominently in many research and commercial projects for several decades. The CMU Navigation Laboratory (Navlab) has built a series of robot cars, SUVs, and buses since 1984. The latest robotic car, Navlab 11, is a robot Jeep Wrangler equipped with a range of sensors for obstacle avoidance, path planning and following, and pedestrian detection [1]. The European Technology Platform on Smart Systems Integration project has reported significant contributions to collision avoidance, fleet management, autonomous cruise control, and cooperative driving [2]. Over the past several years, both Google and Tesla have been commercializing their self-driving platforms [3, 4].

Proponents of AVs argue that the major benefits of driverless cars include less traffic congestion, enhanced mobility for the elderly and the disabled, significant increases in roadway capacity, and reduction in traffic accidents [5, 6]. The claim about the reduction in traffic accidents is typically supported by the argument that since all driverless cars will use the same algorithms, they will act predictably and in unison with respect to each other.

Opponents of driverless cars argue that the widespread adoption of AVs will result not only in major losses of driving jobs, but also will likely lead to loss of privacy and increased risks of hacking attacks and terrorism [7]. Some researchers argue that lack of stress during driving and more productive time on the road may create additional incentives to live even further from cities, which will increase the carbon footprint of motor transportation systems [8].

While we believe that completely autonomous cars may become a reality in the long term, provided that not only technical failures [9, 10] but also social and legal implications [11] of AV adoption are properly addressed, human drivers are, and will remain indispensable in the short and medium terms. Consequently, it is important to seek solutions that enhance their safety. Robust vision-based lane detection is one such enhancement. Specifically, vision-based lane detection modules may gradually become an integral part of autopilots in semi-trucks to improve the drivers' safety

on long, monotonous highway stretches with low or no traffic. Such autopilots will be similar to the ones already in use in aircraft and ships and will keep the human in the loop in that the decision to engage or disengage the autopilot will be under the driver's control.

In this article, an algorithm, called *GreedyHaarSpiker*, is presented for in situ vision-based detection of marked highway lane boundaries on a raspberry pi computer with a raspberry pi camera board. It is assumed that the lane boundaries are marked with white or yellow lines, as is the case on highways in many countries. The computer-camera unit is placed inside a 2016 Jeep Wrangler, next to the windshield, and is powered through a 12V-to-5V car charger. The algorithm is based on the detection of 1D Haar Wavelet spikes [12] in 1D Ordered Haar Wavelet Transforms (HWT) of image rows. The algorithm is implemented in Python 2.7.9 and OpenCV 3.

This article is organized as follows. In Section 2, related work is reviewed. In Section 3, the concept of a 1D Haar Wavelet Spike (1D HWS) is formally developed. Section 4 describes the proposed algorithm and analyzes its pseudocode. In Section 5, the highway experiments are described and analyzed. Findings and conclusions are presented in Section 6.

2 Related Work

Vision-based lane detection has been the focus of many research and development (R&D) projects in the past two decades. Wang et al. [13] propose a B-Snake based lane detection and tracking model for a range of lane structures. An algorithm, called *CHEVP*, is developed for providing initial positions for the B-Snake model. A minimum error method is proposed to determine the control points of the B-Snake model by the image forces on both sides of a lane. Experimental results suggest that the algorithm is robust against noise, shadows, and illumination variations in captured images of marked and unmarked roads.

Kim [14] presents a lane detection and tracking algorithm to detect lane curvatures, lane changes, and splitting lanes. The detected lane markings are grouped into separate left and right lane-boundary hypotheses to handle merging and splitting lanes. The hypotheses are evaluated and grouped with a probabilistic, Markov-style framework.

Hsiao et al. [15] propose an embedded real-time lane departure warning system (LDWS) for daytime and nighttime driving. The LDWS features a lane detection algorithm based on peak finding for feature extraction to detect lane boundaries. Gaussian smoothing and global edge detection are applied to reduce noise in images. The reported lane detection rates were 99.57% during the day and 98.88% at night on a sample of highway images.

Erickson and Landberg [16] proposed a lane detection algorithm that uses Hough lines [17] combined with a parabolic second degree fitting for curvature detection. On the raspberry pi 2 model, the algorithm's performance was found to be inadequate for high speed driving. However, when the object detection is removed from the algorithm, the algorithm meets the real time performance requirements on the raspberry pi 2 model.

Mandlik and Deshmukh [18] have developed a lane departure detection system that uses the OpenCV library [19] to detect vehicle lane departures on the raspberry pi hardware. The algorithm uses the OpenCV implementations of the Canny Edge detector [20] and the Hough Transform [17] to detect straight and curved lanes. The experiments are conducted on a toy vehicle with a USB camera mounted on top of it for sending images of white paper lanes on a black floor surface to a raspberry pi computer powered by a laptop.

The position advocated in this article is similar to the positions advocated in [16] and [18]: to be economically viable and broadly shareable, vision-based lane detection algorithms must be implemented and tested in situ on off-the-shelf low-voltage hardware platforms such as the raspberry pi. The creation of replicable hardware and software solutions will enable citizen science drivers to build, test, and broadly share replicable driver safety enhancements.

3 Haar Wavelet Spikes

The GreedyHaarSpiker algorithm described in Section 4 depends on the concept of the 1D Haar Wavelet Spike developed in this section. In the 1D Haar Wavelet Transform (1D HWT), a signal is a vector in $R^n, n = 2^k, k \in N$. Following the formalization in [21], let $W_a^{(k)}$ be a $2^k \times 2^k$ matrix for computing k scales of the 1D HWT. This matrix can be effectively computed from the n canonical base vectors of R^n. If $x = (x_0, ..., x_{2^k-1})$ is a signal in R^n, then y is the k-scale 1D HWT of x defined in (1).

$$W_a^{(k)} x^T = y \tag{1}$$

The transform of the signal is given in (2), where $a_0^{(0)} = \mu(y)$ and $c_i^{(j)}$ is the coefficient of the i-th basic Haar wavelet at scale j [22]. For example, (3) defines the matrix for computing the 1D HWT in R^2.

$$y^T = (a_0^{(0)}, c_0^{(0)}, c_0^{(1)}, c_1^{(1)}, ..., c_0^{(k-1)}, ..., c_{2^{k-1}-1}^{(k-1)}) \tag{2}$$

$$W_a^{(2)} = \begin{bmatrix} 0.25 & 0.25 & 0.25 & 0.25 \\ 0.25 & 0.25 & -0.25 & -0.25 \\ 0.50 & -0.50 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.50 & -0.50 \end{bmatrix} \tag{3}$$


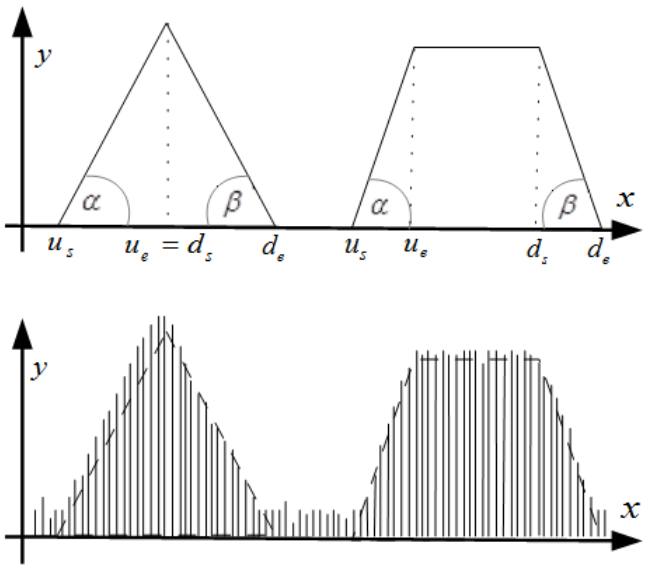

Figure 1: Two types of up-down spikes (above) and the corresponding Haar wavelets at a given scale k from a signal (below).

If the input signal $x = (0, 1, 1, 0)$, then (4) gives the 1D HWT of x computed as $W_a^{(2)} x^T = y$, where $y^T = (0.5, 0, -0.5, 0.5)$.

$$\left[W_a^{(2)}\right] \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.50 \\ 0.00 \\ -0.50 \\ 0.50 \end{bmatrix} \tag{4}$$

It has been theoretically proven that the HWT can detect significant changes in signal values [23]. In this article, we claim that some changes can be characterized as signal spikes [12]. Suppose that there is a finite 1D digital signal. The signal's values may first rise and then fall or they may first fall and then rise. The signal's values may also have a relatively flat segment between the rise and the fall or the fall and the rise. Of course, the signal's values may remain flat for the entire duration of the signal, but a flat signal is not particularly interesting in the sense that it indicates that the underlying phenomenon modeled by the signal does not change.

To model the behavior of 1D digital signals, four types of spikes are postulated: up-down triangle, up-down trapezoid, down-up triangle, and down-up trapezoid. The difference between up-down and down-up spikes is, as their names suggest, the relative positions of the climb and decline segments. In trapezoid spikes, flat segments are always in between the climb and decline segments. One can also view triangle spikes as trapezoid spikes with zero flat segments.

Fig. 1 shows up-down triangle and trapezoid spikes. Fig. 2 shows down-up triangle and down-up trapezoid spikes. In both figures, the lower graphs

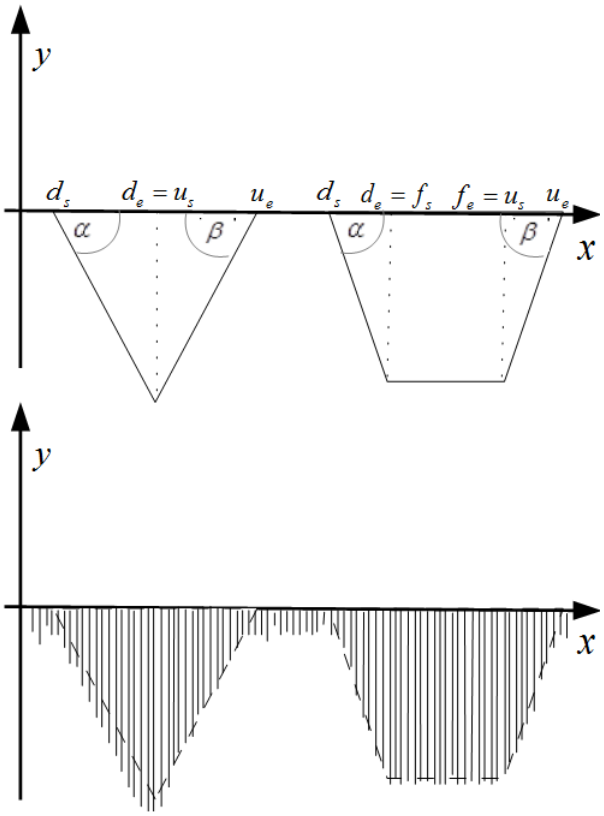

Figure 2: Two types of down-up spikes (above) and the corresponding Haar wavelets at a chosen scale k from a signal (below).

represent the possible values of the corresponding Haar wavelets at a chosen scale k. Up-down spikes describe signals that first increase and then, after an optional flat segment, decrease. Down-up spikes describe signals that first decrease and then, after an optional flat segment, increase.

Let $\mathcal{S}$ be a spike. Then, formally, a spike is a nine element tuple whose elements are real numbers given in (5).

$$\mathcal{S} = (u_s, u_e, \alpha, f_s, f_e, \gamma, d_s, d_e, \beta) \tag{5}$$

The first two elements, u_s and u_e, are the abscissae of the start and end of the spike's climb segment $[u_s, u_e]$, respectively, on which the wavelet coefficients of the 1D HWT increase. If $\omega_{u_s}^{(k)}$ and $\omega_{u_e}^{(k)}$ are the k-th scale wavelet coefficient ordinates at u_s and u_e, respectively, the steepness of the climb, denoted by α, is given in (6).

$$\alpha = tan^{-1}(u_e - u_s, \omega_{u_e}^{(k)} - \omega_{u_s}^{(k)}) \tag{6}$$

As shown in Fig. 1 and Fig. 2, in (5), the flat segments of up-down or down-up spikes, are described by f_s, f_e, and γ, where f_s and f_e in (5) are the abscissae of the start and end of the spike's flat segment, respectively, over which the wavelet coefficients either

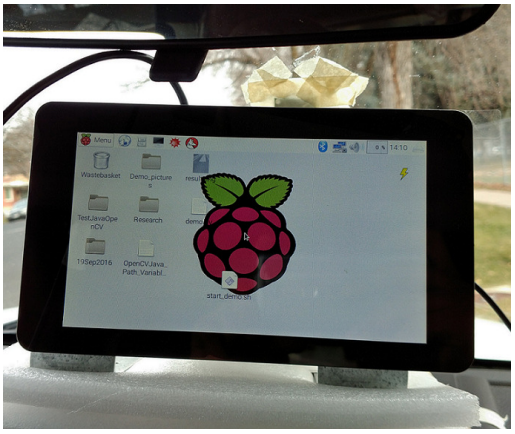

Figure 3: A 7-inch raspberry pi touchscreen display where the output of the algorithm is shown. The display is mounted inside a Jeep Wrangler under the rear view mirror in the middle of the windshield.

Figure 4: A raspberry pi v2 camera, shown by a red arrow, is attached to a small cardboard box. The upper edge of the camera is taped to the windshield. The raspberry pi computer, shown by a green arrow, is attached to the back side of the display shown by a blue arrow.

remain at the same ordinate or have minor ordinate fluctuations.

If $\omega_{f_s}^{(k)}$ and $\omega_{f_e}^{(k)}$ are the k-th scale wavelet coefficients corresponding to f_s and f_e, respectively, the spike's flatness, denoted by γ, is defined in (7).

$$\gamma = tan^{-1}(f_e - f_s, \omega_{f_e}^{(k)} - \omega_{f_s}^{(k)}) \tag{7}$$

The numbers of d_s and d_e in (5) are the abscissae of the start and end, respectively, of the spike's decline segment $[d_s, d_e]$, over which the wavelet coefficients of the 1D HWT decrease.

If $\omega_{d_s}^{(k)}$ and $\omega_{d_e}^{(k)}$ are the k-th scale wavelet coefficient ordinates at d_s and d_e, respectively, the steepness of the decline, denoted by β, is given in (8).

$$\beta = tan^{-1}(d_e - d_s, \omega_{d_e}^{(k)} - \omega_{d_s}^{(k)}) \tag{8}$$

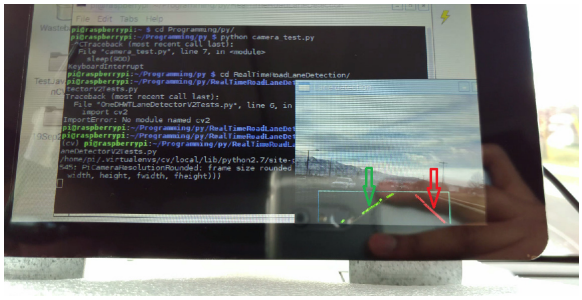

Figure 5: The detected lane boundaries are graphically displayed in the bottom right corner of the display. The green arrow points to the detected left boundary. The red arrow points to the detected right lane boundary.

4 GreedyHaarSpiker: Detection of Lane Boundaries

Figures 3, 4, and 5 show the hardware on which *GreedyHaarSpiker*, the lane boundary detection algorithm described in this article, currently runs. In Fig. 3, a seven-inch raspberry pi touchscreen monitor is shown. As shown in Fig. 4, the monitor is attached to a raspberry pi 3 model B ARMv8 computer with 1GB RAM identified by a green arrow. The computer is attached to the back of the monitor and coupled to a raspberry pi camera board v2. The camera, identified by a red arrow, is attached to a small cardboard box and taped to the windshield for balance. In the future, more stable structures will be designed and deployed.

In Fig. 5, the monitor displays the left and right lane boundaries as they are being detected by the algorithm in real time as the vehicle is driven. The whole system is powered through a 12V-to-5V car charger where the USB power line for the raspberry pi is plugged.

Figure 6: Sample frame from video 1.

Algorithm 1 gives the pseudocode of the procedure *detectLaneBoundaries*. The procedure takes as input a 360×240 PNG image like the one shown in Fig. 6. The ROI in the bottom center of the image, shown as a white rectangle in the center of the image in Fig. 7,

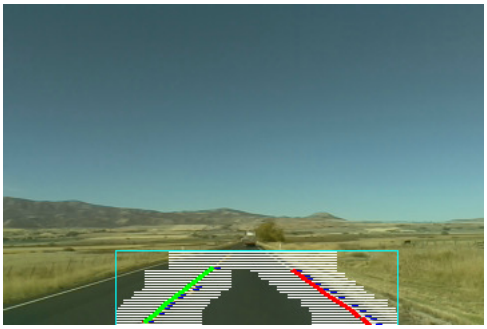

Figure 7: A region of interest with scanlines and detected lane boundaries.

Figure 8: The lane boundaries detected in the frame in Fig. 6. The green line marks the left boundary; the red line marks the right boundary.

is cropped. The cropped ROI is grayscaled, blurred with the 7×7 Gaussian kernel, and thresholded with the Otsu thresholding operator.

The procedure *greedyHaarSpiker*, outlined in Algorithm 2, is applied to the preprocessed ROI. This procedure returns two lists, LPoints and RPoints, of (x, y) tuples. The procedure *fitLine* uses linear regression to fit a line through LPoints and RPoints. The lines are filtered by slope to reduce false positives. The slope thresholds for the left boundary are from -60 to -30; the slope thresholds for the right boundary are from 30 to 60. If the line through LPoints passes the left slope threshold filter, it is taken to be the left boundary of the vehicle's lane. If the line through RPoints passes the right slope treshold filter, it is taken to be the right boundary of the vehicle's lane.

Algorithm 1 *detectLaneBoundaries*(Img)

```
ROI ← cropROI(Img);
ROI ← convertToGrayscale(ROI);
ROI ← gaussianBlur(ROI);
ROI ← thresholdOTSU(ROI);
LPoints, RPoints ← greedyHaarSpiker(ROI);
LeftLaneBoundary ← fitLine(ROI, LPoints);
RightLaneBoundary ← fitLine(ROI, RPoints);
return LeftLaneBoundary, RightLaneBoundary;
```

Algorithm 2 *greedyHaarSpiker*(ROI, s_r, e_r, Δ)

```
LLPoints ← [];
RPoints ← [];
LSpike ← NULL;
RSpike ← NULL;
r ← s_r;
while r ≥ e_r do
  LLine ← getLeftScanLine(ROI, r, LSpike);
  RLine ← getRightScanLine(ROI, r, RSpike);
  LHWT ← ordHWT(LLine);
  RHWT ← ordHWT(RLine);
  LSpike ← detectSpike(LHWT);
  RSpike ← detectSpike(RHWT);
  if LSpike ≠ NULL then
    LPoints.add(LSpike.getMidPointOfClimb());
  end if
  if RSpike ≠ NULL then
    RPoints.add(RSpike.getMidPointOfClimb());
  end if
  r ← r + Δ;
end while
```

The procedure *greedyHaarSpiker* in Algorithm 2 takes a preprocessed ROI and three integer parameters s_r, e_r, and Δ. The parameters s_r and e_r ($s_r \geq e_r$) specify the start and end rows, respectively, in the ROI where the spikes are detected. The parameter Δ specifies a step value, a small negative integer, for generating the exact row numbers where spikes are detected. For example, if the algorithm is to detect spikes in the row range $[50, 40]$, i.e., $s_r = 50$ and $e_r = 40$, with $\Delta = -2$, the sequence of rows that will be considered is $(50, 48, 46, 44, 42, 40)$. Note that the spike detection starts from the lower rows that are closest to the vehicle and moves up to the the rows that are further away from the vehicle.

The variables LPoints and RPoints contain the (x, y) tuples returned to *detectLanes* after linear regression line fitting. The variables LSpike and RSpike contain two spikes detected in the ordered HWTs of the left and right scanlines, respectively.

In the **while**-loop of *greedyHaarSpiker*, two scanlines, LLine and RLine, of 64 pixels each are chosen on the left and right sides of the ROI in row r. The scanline's length, i.e., 64, can be changed through a global variable but it has to be equal to an integral power of 2. A value of 64 was experimentally found to result in optimal performance.

If the value of LSpike is NULL, the left scanline starts at column 0. Similarly, if the value of RSpike is NULL, the right scanline starts at column $w - 1$, where w is the width of the ROI, which, in the current implementation, is equal to 200. If the value of LSpike is not NULL, which means that a spike was detected in the previous row, the left scanline, saved in the LLine variable, is centered on the middle of the two ordinates of the detected spike's climb segment, i.e.,

the ordinates of u_s and u_e in equation 5. The flat and down segments are currently not taken into account in the algorithm. The right scanline is detected and saved in RLine in the same way except that the spike saved in RSpike is used. In Fig. 7, the scanlines are shown as horizontal white lines on the left and right sides of each row. As row number r apporoaches the upper boundary of the ROI (i.e., e_r) the gap between the left and right scan lines becomes smaller.

The procedure *detectSpike* in the **while**-loop of the procedure *greedyHaarSpiker* uses thresholds for the angles of the climb, flat, and decline spike segments, i.e., α, γ, β in (5), and returns the leftmost spike that clears the thresholds. In the current implementation, $\alpha = \beta = 60°$ and $\gamma = 5°$. In other words, the spikes whose climb or decline angles are less than $60°$ are filtered out, and flat segments are detected so long as consecutive wave coefficients fluctuate within $\pm 5°$ of 0. This algorithm is greedy in that it always returns exactly one leftmost spike in each left scanline and exactly one leftmost spike in each right scanline. All other spikes are ignored. If no spikes clear the angle thresholds, the value of NULL is returned.

When **while**-loop of *greedyHaarSpiker* finishes, the lists LPoints and RPoints contain (x, y) tuples representing the mid points of the climb segments of spikes detected in the left and right scanlines in each of the processed rows. These points are used by the procedure *fitLine* in *detectLanes* to fit lines through them. The lines identify the left and right lane boundaries, as shown in Fig. 8.

5 Experiments

The images for the experiments were captured with the hardware shown in Figures 3, 4, and 5. To obtain experimental video data for daytime driving, the author drove a 2016 Jeep Wrangler on two different days in September (run 1) and November 2016 (run 2) at a speed of 55-60 miles per hour on Route 30, a two-lane Northern Utah highway with marked lane boundaries. On each run, the raspberry pi camera unit was turned on to record the video and save it on the raspberry pi's sdcard. The first run was on a sunny day with clear skies and good visibility. The second drive was on a cloudy day with light rain. To obtain experimental video data for driving on snowy roads and night driving, the author drove the same vehicle on the same highway and at the same speed on a day in January 2017 after a heavy snowfall (run 3) and on the same day after sunset (run 4).

The first three runs were approximately 35 miles long. The fourth run was approximately 20 miles long. The video from each run was segmented into frames and a sample of 360 x 240 consecutive PNG frames was selected from it. Samples 1, 2, and 3 had 1,000 frames each selected from the videos recorded in runs 1, 2, and 3, respectively; sample 4, selected from the run 4 video, included 775 frames.

Table 1: Lane boundary detection accuracy

SN	NB = 2 (%)	NB $\geq$ 1 (%)	FP (%)
1	61.90	91.20	1.60
2	34.10	77.40	2.70
3	16.90	64.10	8.30
4	15.74	57.03	11.48

The evaluation of the algorithm's performance was done manually by two human evaluators who compared the lane boundaries drawn in each image by the algorithm with the actual lane boundaries in the same image. Each image thus evaluated was placed into one of the three accuracy categories: both boundaries detected, at least one boundary detected, and no boundary detected. An actual boundary was considered detected accurately if the boundary line drawn by the algorithm was exactly aligned with the actual boundary.

Table 1 shows the accuracy results on all four image samples. The column SN, which stands for sample number, lists the four sample numbers. The second column, NB = 2, shows the percentage of frames where the number of detected boundaries (NB) is exactly 2, i.e., both boundaries are detected. The third column, NB $\geq$ 1, shows the percentage of frames where at least one boundary was accurately detected. The fourth column gives the percentage of false positives (FP) in each sample.

In sample 1, both boundaries were accurately detected in 61.9% of the frames and at least one lane boundary was detected in 91.20% of the images. The percentage of false positives in sample 1 is 1.6%. The detection results in sample 2 were 34.10% for both boundaries and 77.40% for at least one lane boundary. The percentage of false positives in sample 2 was 2.70%. In sample 3, taken on a winter day, the percentage of both lanes detected dropped to 16.90% and the percentage of at least one lane detected decreased to 64.10%. The percentage of false positives in samle 3 increased to 8.30%. Finally, in sample 4, the lane identification performance was the worst. Specifically, the percentage of both lanes detected was 15.74%, the percentage of at least one lane detected was 57.03%, and the percentage of false positives increased to 11.48%.

Fig. 9 illustrates an inaccurate boundary detection and a false positive from sample 1. While the left lane boundary, denoted by a small bright green line, is accurately aligned with a real boundary, it is aligned with the boundary of the opposite lane. This misalignment shows a problem with the greedy approach in that the algorithm always chooses the leftmost spike in each scanline. The red line, almost perpendicular to the bright green line, is a false positive.

Fig. 10 shows two frames from run 1 taken on a sunny day. The left image shows both boundaries detected accurately. The right image shows only the


Figure 9: Run 1: a short green line is inaccurately aligned with a wrong boundary; a red line is a false positive.

Figure 10: Run 1: both lanes recognized (left); only the left boundary recognized (right).

Figure 11: Run 2: both lanes recognized (left); no left boundary recognized and a false right boundary (right).

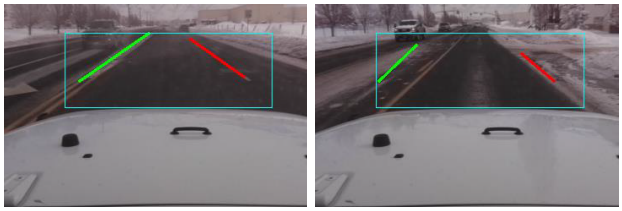

Figure 12: Run 3: both lanes recognized (left); a false left boundary and a correct right boundary (right).

Figure 13: Run 4: both lanes recognized (left); neither lane recognized (right).

left boundary (green line) detected accurately while the right boundary is not detected at all. The failure to detect the right boundary was caused by the shadow of a semitruck in the opposite lane.

Fig. 11 shows two frames from run 2 taken on a cloudy day. The left image shows both lanes accurately recognized. In the right image, the left boundary is not detected and the right boundary is detected inaccurately. This is also a case of a false positive.

Fig. 12 shows two frames from run 3 taken after a heavy snowfall. The left image shows both lanes accurately recognized. In the right image, the left boundary is detected inaccurately and the right lane is accurately recognized. There were many instances of detection failures because the lanes were covered by snow.

Fig. 13 shows two frames from run 4 taken at night after a heavy snowfall. The left image shows both boundaries detected accurately. It is interesting to note that the lane detection recognition was better when there were cars in the opposite lane with their lights turned on. The right image shows a frame where neither boundary was detected.

6 Conclusions

In this article, an algorithm was presented for in situ vision-based detection of highway lane boundaries on a raspberry pi computer coupled to a raspberry pi camera. The algorithm, called *GreedyHaarSpiker*, is based on the detection of 1D Haar Wavelet spikes in 1D Ordered Haar Wavelet Transforms of image rows.

The position advocated in this article is that, in order to be economically viable and broadly shareable, vision-based lane detection algorithms should be implemented and tested in situ on off-the-shelf low-voltage hardware platforms such as the raspberry pi. The creation of replicable hardware and software solutions will enable citizen science drivers to build, test, and broadly share replicable driver safety enhancements.

To address the lane detection problems described in Section 5, several improvements are being considered. Recall that, as explained in Section 4, the flat and down segments are currently not taken into account in the algorithm. Thus, the first improvement

is to use not just the climb segment of each detected spike but also the flat and decline segments when computing the 2D points of a potential line either on the left or on the right. The second improvement is to use a different curve fitting algorithm, e.g., a high order polynomial, to find the best line that fits a set of points. A potential drop in the number of frames processed per second may be compensated by more accurate lane boundary detection. The third improvement is to add geometrical constraints to reduce the number of false positive.

Acknowledgements

The author is grateful to Vikas Reddy Sudini for helping him evaluate the algorithm's performance.

References

[1] S. Thrun. Toward Robotic Cars. *Communications of the ACM*, 53(4):99–106, 2010.

[2] J. Dokic, B. Müller, and G. Meyer. *European Roadmap Smart Systems for Automated Driving.* European Technology Platform on Smart System Integration, Berlin, Germany, 2015.

[3] T. Simonite. Data shows google's robot cars are smoother, safer drivers than you or i. *MIT Technology Review*, Oct., 2013.

[4] G. Nelson. Tesla beams down 'autopilot' mode to model s. *Automative News*, Oct. 14, 2015.

[5] C. Mui. Will the google car force a choice between lives and jobs? *Forbes*, Dec., 2013.

[6] T. Lassa. The beginning of the end of driving. *Motor Trend*, Jan., 2013.

[7] O. Miller. Robotic cars and their new crime paradigms. *LinkedIn Pulse*, Sept. 3, 2013.

[8] M. Ufberg. Whoops: The self-driving tesla may make us love urban sprawl again. *Wired*, Oct. 10, 2015.

[9] D. Yadron and D. Tynan. Tesla driver dies in first fatal crash while using autopilot mode. *The Guardian*, Jul. 1, 2016.

[10] V. Mathur. Google autonomous car experiences another crash. *Government Technology*, Jul. 17, 2015.

[11] J. Boeglin. The costs of self-driving cars: reconciling freedom and privacy with tort liability in autonomous vehicle regulation. *Yale Journal of Law and Technology*, 17(1):Article 4, 2015.

[12] V. Kulyukin and V. R. Sudini. Real-Time Vision-Based Lane Detection on Raspberry Pi with 1D Haar Wavelet Spikes. In *Lecture Notes in Engineering and Computer Science: Proceedings of the International MultiConference of Engineers and Computer Scientists*, pages 75–80, Hong Kong, China, March 2017.

[13] Y. Wang, E. Teoha, and D. Shen. Lane detection and tracking using b-snake. *Image and Vision Computing*, 22:269–280, 2008.

[14] Z. Wang. Robust lane detection and tracking in challenging scenarios. *IEEE Trans. on Intelligent Transportation Systems*, 9(1):16–26, 2008.

[15] P. Hsiao, C. Yeh, S. Huang, and L. Fu. A portable vision-based real-time lane departure warning system: day and night. *IEEE Trans. on Vehicular Technology*, 58(4):2089–2094, 2009.

[16] J. Eriksson and J. Landberg. *Lane departure warning and object detection through sensor fusion of cellphone data.* Master's thesis in Applied Physics and Complex Adaptive Systems, Department of Applied Mechanics, Chalmers University of Technology. Göteborg, Sweden, 2015.

[17] R.O. Duda and P.E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Comm. ACM*, 15:11–15, 1972.

[18] P. Mandlik and A. Deshmukh. Raspberry-pi based real time lane departure warning system using image processing. *International Journal of Engineering Research and Technology*, 5(6):755–762, 2016.

[19] R. Laganiere. *OpenCV 2 Computer Vision Application Programming Cookbook.* Packt Publishing LTD, 2011.

[20] J.F. Canny. A Computational approach to edge detection. *IEEE Trans. on Pat. Anal. And Mach. Intel.*, 8:679–688, 1986.

[21] A. Jensen and A. Cour-Harbo. *Ripples in mathematics: the discrete wavelet transform.* New York: Springer, 2011.

[22] Y. Nievergelt. *Wavelets made easy.* Boston: Birkhäser, 2001.

[23] S. Mallat and W. Hwang. Singularity detection and processing with wavelets. *IEEE Trans. on Information Theory*, 38(2):617–643, 1992.

Biographies

Vladimir A. Kulyukin is an Associate Professor of Computer Science at Utah State University. He holds a Ph.D. in Computer Science from the University of Chicago that he received in 1998. His research interests include AI, computer vision, and sensor fusion.